CS270 Recitation 4
C Structures

Make a subdirectory called **R4** for the recitation, all files should reside in this subdirectory. Copy the following files to the R4 directory:

https://www.cs.colostate.edu/~cs270/.Spring17/recitations/R4/src/struct.h
https://www.cs.colostate.edu/~cs270/.Spring17/recitations/R4/src/struct.c
https://www.cs.colostate.edu/~cs270/.Spring17/recitations/R4/src/main.c
https://www.cs.colostate.edu/~cs270/.Spring17/recitations/R4/src/Makefile
https://www.cs.colostate.edu/~cs270/.Spring17/recitations/R4/src/class.txt

Then, type **make** to compile the program.

In this recitation, we will create a program to read student information from standard input, store it into a class roster data structure, and display the students on standard output.

When we talk about standard input, we usually think of the user entering information from the keyboard. However, when you're testing a program, entering the same information over and over can become tedious. We will automate user input by using a feature of the shell called **input redirection**. The user input will come from a file. We have provided an example file (**class.txt**). The format is as follows (the stuff shown in green is not part of the file). In general, the number of students in the file is not fixed.

```
3        <- Number of students in the class
John     <- First student's first name (string)
52.5     <- First student's quality points (float)
15       <- First student's number of credits (integer)
Jane     <- Second student's first name (string)
53.2     <- Second student's quality points (float)
14       <- Second student's number of credits (integer)
Johnny   <- Third student's first name (string)
35.7     <- Third student's quality points (float)
17       <- Third student's number of credits (integer)
```

In order to make our C program read from this file as if it was the standard input, we will use the following command (< is the input redirection operator):

```
./R4 < class.txt
```

Now that you're familiar with the format of the file, let's get to coding:

1. In the **struct.h** file, complete the **Student** structure so that it contains the following three members:

    a. **firstName:** a string (**array of characters**) whose size is 80 characters.

b. **qualityPoints:** a **float**.

c. **numCredits:** an **int**.

This structure simply encapsulates a student.

2. In the **struct.h** file, declare a structure using **typedef**. The name of the new type should be **ClassRoster**. The structure should contain the following members:

   a. **numStudents:** an **int**.

   b. **students:** a **Student** ** (yes, a double pointer).

   At this point, try to recompile your program to make sure you don't have errors.

3. In the main function (in **main.c**) do the following:

   a. Read the number of students from the standard input using the **scanf** function. To use this function, you must provide a format string. The call would look something like this:

      ```
      scanf("%d", address of variable);
      ```

      The **%d** indicates that we want to read an integer. Replace *`address of variable`* with an expression that gets the address of the variable where you want to store the number of students. By this point in the class, you should understand why we have to pass the address of the variable to **scanf**. If you don't, ask your TA.

   b. Declare a variable of type **ClassRoster** named **roster**. This allocates space for a **ClassRoster** structure statically (as opposed to dynamically). Then, initialize the two members of this structure as follows:

      i. The **numStudents** member should be initialized to the number you read in (a).

      ii. The **students** member should be initialized to point to a dynamically allocated array of **Student pointers** (not an array of Students!). The number of elements in the array should be the number of students in the class. You want all the elements in the array to be initialized to NULL (or 0). With this in mind, should you use **malloc()** or **calloc()**?

      At this point, you should understand why the type of the **students** member is a double pointer. If not, ask your TA.

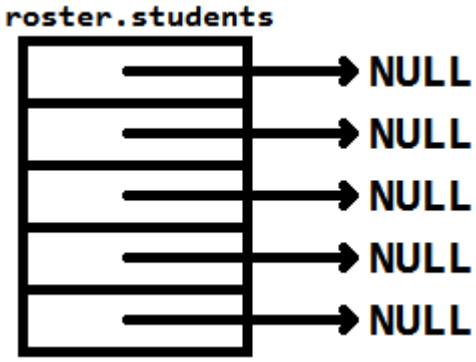
   c. Write a **for** loop that iterates as many times as dictated by the **numStudents** member of the **roster** structure. The goal is to iterate through the **students** array to read students from the standard input and store them in the array. In each iteration, call the **readStudentAndEnroll()** function (this function is declared in **struct.h**). Pass the **address** of the current element of the array. Based on this, you should understand why this function takes a double pointer.

d. Write another separate **for** loop that iterates as many times as dictated by the **numStudents** member of the **roster** structure. The goal is to iterate through the **students** array and display each element in the standard output. In each iteration, call the **displayStudent()** function (this function is declared in **struct.h**). Notice that this function takes a **Student** (not a **Student** *). You must call it accordingly, don't change the declaration! After calling **displayStudent()**, call **free()** to free the memory associated with this student.

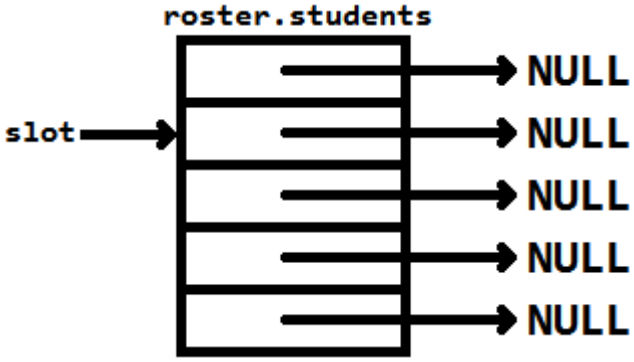
e. After the last **for** loop, call **free()** to free the memory associated with the **students** array. Why do we not need to call **free()** to free roster?

4. Try to compile your program. At this point, you shouldn't have warnings or errors.

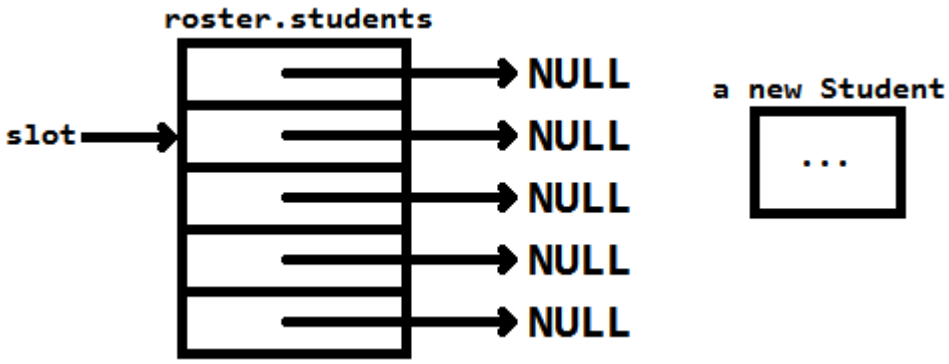
5. Now, we need to complete the **readStudentAndEnroll()** function. It may be tricky to understand why this function takes a double pointer. First, remember that in the main function, we created an array of **Student pointers** initially pointing to NULL. We can visualize this array as follows:



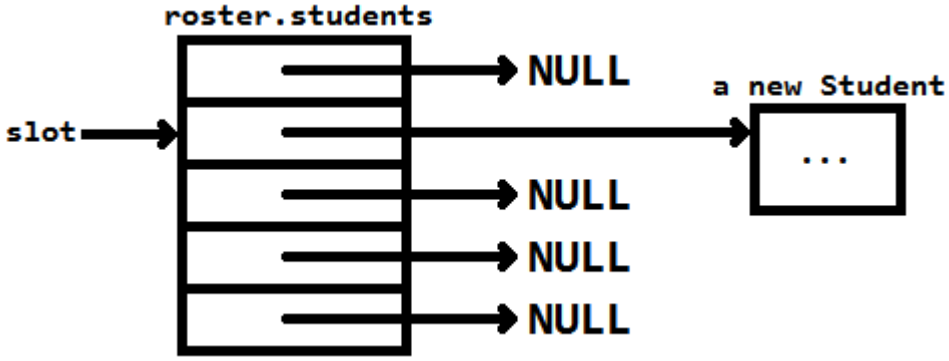
Then, we created a loop to call **readStudentAndEnroll()** to pass the address of each element of the array. Since every element of the array is a **Student pointer**, and this function accepts the address of such an element (we'll see why shortly), the type of the parameter must be a **pointer to a Student pointer** (or **Student** **). Suppose we are currently dealing with the second element of the array. That means that the **slot** argument of the **readStudentAndEnroll()** function is a pointer to the second pointer of the array:



So, why do we accept the address of a pointer in the first place? This function will create a new **Student** dynamically and will populate it with some information:



Then, we want to make the current element in the array point to the new student. As we saw in a previous recitation, if a function wants to change the original argument, we must pass it the address of what we want it to change. So, if we want **readStudentAndEnroll()** to change what the second element points to, we'd better pass the address of that pointer. Hence, we end up with a double pointer. That way we can use the **slot** argument to achieve this:



With all this in mind, here's what you should do in this function:

a. Dynamically allocate space for a new **Student**. We will initialize the members shortly. Do you need a **malloc()** or a **calloc()**?

b. Read the next line from standard input using **scanf**, which should be the first name of the student. The first name should be stored in the **firstName** member of the student you just created.

c. Read the next line from standard input using **scanf**, which should be the quality points of the student. The quality points should be stored in the **qualityPoints** member of the student you just created.

d. Read the next line from standard input using **scanf**, which should be the number of credits of the student. The number of credits should be stored in the **numCredits** member of the student you just created.

e. Dereference the **slot** argument to make the original pointer point to the student that we created. If you can come up with this one line on your own, you're on your way to becoming a pointer master.

6. Compile your program and resolve any errors and warnings you find. You may have to include a missing header file to resolve some of the warnings.

7. Finally, we need to complete the **displayStudent()** function. This function will calculate the GPA for a student and output the student's name and the GPA. Here are the steps:

   a. Calculate the GPA (a variable of type **float**) of the student using the following formula:

   ```
   GPA = quality points / number of credits
   ```

   b. Print a line of output with the student's name and the GPA in the following format:

   ```
   John,3.50
   ```

   The GPA needs to have 2 decimal places. Don't forget the newline character at the end.

8. Compile your program and resolve any warnings or errors you may have.

9. Now you can run the program. Here's our output using the provided **class.txt** file:

   ```
   John,3.50
   Jane,3.80
   Johnny,2.10
   ```

   Feel free to test using your own text files. You should also run your program with **Valgrind** to check for errors and memory leaks:

   ```
   valgrind --leak-check=yes ./R4 < class.txt
   ```

   In the output, you should see the following two lines:

   ```
   All heap blocks were freed -- no leaks are possible
   ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
   ```

10. Once you're comfortable with your program, type **make package**. This will generate a file named **R4.tar** which contains the **struct.h**, **struct.c**, and **main.c** files. Submit this tar file to the Checkin tab and wait for the testing results. Your grade for this recitation will be what you get in the auto-grader. You may submit as many times as needed until the deadline.